

1. A Detailed Server-side Overview

1. Bcfg2 Server
2. Bcfg2 Abstractions
 1. Bcfg2 Elements
 2. Bcfg2 Bundles
 3. Bcfg2 Groups
 4. Bcfg2 Profiles
 5. Bcfg2 Generators
3. Bcfg2 Repository
 1. Metadata/ Directory
 1. Metadata/groups.xml File
 2. Metadata/clients.xml File
 2. Bundler/ Directory
 3. Pkgmgr/ Directory
 4. Svcmgr/ Directory
 5. Rules/ Directory
 6. Base/ Directory
 7. Cfg/ Directory
 1. Specialized Configuration Files
 2. Deltas
 1. Cat Files
 3. .info Files
4. Bcfg2 Reports
 1. Report Configuration
 2. Report Scripts
 3. Report Interface

A Detailed Server-side Overview

TracNav menu

- Getting Started
- Download
- Install
- Help
- News
- Contribute
- Publications
- Testimonials
- Sites Using Bcfg2

A typical install of Bcfg2 involves two parts: the server, which hands out configuration information; and the client, which consumes it. The server-side part is by far the more complex of the two. Here we present a detailed look at the parts of a typical server install.

Bcfg2 Server

Each deployment will have a central Bcfg2 server. This machine runs the Bcfg2 server daemon, keep the Bcfg2 configuration repository, and serves package data when needed. The server uses an SSL-encrypted XML-RPC channel for its communication, meaning passing sensitive information like SSH host keys and `passwd` files can be passed safely across the wire. The Bcfg2 server and client get the connection information from the `/etc/bcfg2.conf` configuration file. An example `/etc/bcfg2.conf` looks like:

```
[server]
repository = /var/db/bcfg
structures = Bundler,Base
generators = SSHbase,Cfg,Pkgmgr,Svcmgr,TCheetah
metadata = /var/db/bcfg/etc

[statistics]
domainlist = 'example.com'
sendmailpath = /usr/sbin/sendmail

[communication]
protocol = xmlrpc/ssl
password = verysecret
key = /usr/share/ssl/bcfg2/www-key.pem

[components]
bcfg2 = https://www.example.com:6789
```

This file should be owned by user and group root and readable only by user root.

Bcfg2 Abstractions

Bcfg2 Elements

Host specifications are created through a modular process. At the base of a specification are *configuration elements* which generally correspond to the filesystem and service objects that a particular host needs. The following are all valid configuration elements:

ConfigFile

A config file

Directory

A directory that should be present

Package

A package for the given system (RPM, Deb, ebuild, etc)

Permissions

The Permissions a POSIX element should have

Service

A service to start or stop on boot (usually kept in `/etc/init.d`)

SymLink

A symbolic link that should be present

Bcfg2 Bundles

Bundles are the simplest reusable configuration blocks. A Bcfg2 Bundle is an XML-formatted collection of configuration elements that are tightly related. An example Bundle for `syslog` might look like:

```
<Bundle name="syslog" version="2.0">
  <Group name="rhel">
    <ConfigFile name="/etc/syslog.conf"/>
    <Package name="sysklogd"/>
    <Service name="syslog"/>
  </Group>
</Bundle>
```

Here we can see that for the Group `rhel`, the `syslog` Bundle contains a `ConfigFile` element `/etc/syslog.conf`, a `Package` element `sysklogd`, and a `Service` element `syslog`. This defines a relationship between all these elements, so that if one of these changes, examination of all the elements should be taken more closely. For example, an update to the `/etc/syslog.conf` file will automatically trigger the Bcfg2 client to restart the `syslog` service. Similarly, an update to the `sysklogd` package will cause the client to re-check `/etc/syslog.conf` to make sure it wasn't clobbered by the new package along with the automatic restart of the `syslog` service.

Bundles should be fairly discrete and simple and their relationship should be tight. We wouldn't want to put the `sshd` service in the `syslog` Bundle even though `sshd` does use `syslog`. Restarting `sshd` when `/etc/syslog.conf` changes or when a new `sysklogd` package comes out doesn't make sense.

Bcfg2 Groups

Groups are the next abstraction up from Bundles. Groups can contain Bundles, other Groups, or simply nothing at all. An example Group for a generic server might look like:

```
<Group profile='false' name='server'>
  <Bundle name='kerberos-client' />
  <Group name='backup-client' />
  <Group name='base' />
  <Group name='ldap-client' />
  <Group name='serial-console' />
</Group>
```

Here we see that the `server` group is actually made up of a bundle and several other groups. It will inherit any bundles included in these groups, as well as the explicit bundles it includes. For example, we might find it makes sense to include our earlier `syslog` bundle in the `base` group so that it can be shared among all of our machines:

```
<Group name='base'>
  <Bundle name='syslog' />
</Group>
```

By then including the `base` group in our above `server` group, any machine associated with the `server` group will gain all of the configuration elements provided by the `syslog` bundle.

Groups are used throughout Bcfg2 to give control over configuration elements as a whole. They can be used in Bundles to help distinguish which elements a host gets, by the `Pkgmgr` plugin to specify which version of a package a host should install, by the `Svcmgr` plugin to control if a host should activate or deactivate a particular service, and by the `Cfg` plugin to finely control specific versions of files to be given to hosts.

Bcfg2 Profiles

The highest level of abstraction is that of Profiles. A Profile is a special Group and is then mapped to a host. A host can have one and only one profile mapped to it. Note that a Profile can also contain other Profiles, but these Profiles aren't mapped to the host, they are simply associated with the host as a Group. A mail-server Profile might look like:

```
<Group profile='true' public='false' name='mail-server'>
  <Bundle name='mail-server' />
  <Bundle name='mailman-server' />
  <Group name='apache-server' />
  <Group name='rhel-as-4-x86' />
  <Group name='nfs-client' />
  <Group name='server' />
</Group>
```

Here we can see that a mail-server is actually made up of a couple of Bundles and a few Groups as well.

Bcfg2 Generators

Generators are a special case for Bcfg2. A Generator is a piece of code that is run to generate Bcfg2 Elements automatically for a host. To explain this the best way it's best to look at an example.

Integrated Unix-like environments often run SSH on all of their machines. One of the most tedious processes involved with a large group of SSH-running machines is keeping track of SSH host keys: when a new machine is built, a new set of host keys is created, and when an existing machine is rebuilt, it is desirable to keep the existing set of keys for the new instance. It is often the case that a global `ssh_known_hosts` file must be maintained with keys for all machines in the environment. While this can all be done by hand, it can easily be automated. Enter Generators.

A Generator can take care of a particular ConfigFile element. Any time this element is requested by the client, the server dynamically generates it either by crunching data and creating new information or by reading a file off of disk and passes it down to the client for installation. For example, the SSHBase generator, which solves the above problem, manages a directory of SSH keys on disk. When a new machine is built, the generator creates a new set of SSH host keys for it, saves them to disk, and passes them down to the client for installation. When an existing client asks for its host keys, the generator merely reads the existing keys from disk and passes them along. Finally, when a client asks for a `known_hosts` file, the generator creates a copy from the current set of keys it knows about and sends that down to the client.

As an added exercise, let's look at what happens when a machine is compromised. Two things need to be done with respect to SSH keys: new keys need to be created and installed, and the `known_hosts` files on all other machines need to be updated. With the SSHBase generator, this is achieved by simply removing the host's keys from the SSHBase directory before you rebuild it. When the machine rebuilds and runs the Bcfg2 client, it will automatically get a brand-new key, and that key will automatically be included in the `known_hosts` file when the rest of your hosts run next. Simple!

Bcfg2 Repository

The Bcfg2 repository is just a collection of directories and files that direct the Bcfg2 server how to build configurations for hosts. We assume the repository is kept in `/var/db/bcfg`. Within the central Bcfg2 repository there are several directories that contain configuration parameters for establishing a particular configuration for a host.

Metadata/ Directory

The Metadata/ directory contains several configuration files that dictate how to start generating a configuration for a host.

Metadata/groups.xml File

The groups.xml contains Group and Profile definitions. Here's a very basic Metadata/groups.xml:

```
<Groups version='3.0'>
  ...
  <Group profile='true' public='false' name='mail-server'>
    <Bundle name='mail-server' />
    <Bundle name='mailman-server' />
    <Group name='apache-server' />
    <Group name='rhel-as-4-x86' />
    <Group name='nfs-client' />
    <Group name='server' />
  </Group>
  ...
</Groups>
```

Metadata/clients.xml File

The Metadata/clients.xml file contains the mappings of Profiles to clients. A sample from this file is:

```
<Clients version="3.0">
  <Client profile="backup-server" pingable="Y" pingtime="0"
name="backup.example.com"/>
  <Client profile="console-server" pingable="Y" pingtime="0"
name="con.example.com"/>
  <Client profile="kerberos-master" pingable="Y" pingtime="0"
name="kdc.example.com"/>
  <Client profile="mail-server" pingable="Y" pingtime="0"
name="mail.example.com"/>
  ...
</Clients>
```

Bundler/ Directory

The Bundler/ directory is where all the Bcfg2 bundle XML files are kept. These are the files described above.

Pkgmgr/ Directory

The Pkgmgr/ directory keeps the XML files that define what packages are available for a host or image and where to find those packages. Here's an example of the files:

```
$ ls Pkgmgr/
centos-4-noarch-updates.xml
centos-4-x86-updates.xml
centos-4-x86.xml
backup.example.com.xml
fedora-core-4-noarch-updates.xml
fedora-core-4-x86-updates.xml
fedora-core-4-x86.xml
```

```

rhel-as-4-noarch-updates.xml
rhel-as-4-x86-updates.xml
rhel-as-4-x86.xml
rhel-es-4-noarch-updates.xml
rhel-es-4-x86-updates.xml
rhel-es-4-x86.xml
rhel-ws-4-noarch-updates.xml
rhel-ws-4-x86_64-updates.xml
rhel-ws-4-x86_64.xml
rhel-ws-4-x86-updates.xml
rhel-ws-4-x86.xml

```

Here we can see a pattern of files. Looking at the names of these files you can make a pretty good guess as to what they contain.

Let's take a look at what the contents of one of these files, `rhel-as-4-x86.xml`, might look like:

```

<PackageList uri='http://www.example.com/media/rhel-as-4/x86/install/
RedHat/RPMS' type='rpm' priority='0'>
  <Group name='rhel-as-4-x86'>
    <Package file='4Suite-1.0-3.i386.rpm' />
    <Package file='Canna-3.7p3-7.EL4.i386.rpm' />
    <Package file='Canna-devel-3.7p3-7.EL4.i386.rpm' />
    <Package file='Canna-libs-3.7p3-7.EL4.i386.rpm' />
    <Package file='ElectricFence-2.2.2-19.i386.rpm' />
    <Package file='FreeWnn-1.10pl020-5.i386.rpm' />
    <Package file='FreeWnn-devel-1.10pl020-5.i386.rpm' />
    <Package file='FreeWnn-libs-1.10pl020-5.i386.rpm' />
    ...
  </Group>
</PackageList>

```

Here you can see that the data is encapsulated in a `PackageList` which describes the URI of the files described, the type of package, and a priority of the files. The priority is used to decide which specific file to use when there are multiple files that could be used for a particular host. The highest priority file is the one that is used. Using this system, you can have a file that contains all the Packages from the original installation, `rhel-as-x86.xml` in our case, and then create a new file that contains updates that are made available afterwards, `rhel-as-x86-updates.xml` and `rhel-as-noarch-updates.xml` in our case. You simply make the priority of the update Packages higher and they will be used instead of the original installation Packages.

The names of the XML files have no special meaning to `Bcfg2`; they are simply named so it's easy for the admin to know what the contents hold. All Packages could be kept in a single file if you so desired. `Bcfg2` simply uses the Groups in the files and priorities to determine how to assign Packages to a host. You may notice the file `backup.example.com.xml`. Its Packages have a higher priority than the update Packages. This is because that particular host requires special Packages that are older than the ones available in the updates.

Svcmgr/ Directory

The `Svcmgr` directory is responsible for defining the state of Services. Like the `Pkgmgr` directory, the `Svcmgr` directory can contain multiple files to allow the administrator to divide the definitions of services up. Here's a simple example:

```

<Services priority='0'>
  <Service name="kudzu" status="off"/>
  <Group name="mail-server">
    <Service name="postfix" status="on"/>
  </Group>
</Services>

```

```

    </Group>
    ...
</Services>

```

Rules/ Directory

The Rules directory is responsible for defining the details about the Directory, Permissions and SymLink elements. In a future release the Svcmgr directory will be deprecated and Service elements will be defined here as well. Like the Pkgmgr directory, the Rules directory can contain multiple files to allow the administrator to divide the definitions up:

```

$ ls Rules/
directories.xml
permissions.xml
symlinks.xml

```

Here's a simple example from the `directories.xml` file:

```

<Rules priority='0'>
...
  <Group name='base'>
    <Directory name='/mnt' owner='root' group='root' perms='0755' />
    <Directory name='/var/account/rotated-logs' owner='root'
group='root' perms='0755' />
    <Directory name='/var/log/rotated-logs' owner='root' group='root'
perms='0755' />
    <Directory name='/var/log/cups/rotated-logs' owner='root'
group='root' perms='0755' />
    <Directory name='/var/log/ppp/rotated-logs' owner='root'
group='root' perms='0755' />
  </Group>
...
</Rules>

```

Notice that the files in this directories can have priorities just like the Pkgmgr directory.

Base/ Directory

The Base directory is responsible for defining the base config for all hosts of a given Group. The majority of the elements are Packages, but you can have ConfigFiles, Directories, Services and SymLinks as well. Here's an example:

```

<Base >
  <Group name='rhel-as-4-x86'>
    <ConfigFile name='/etc/group' />
    <ConfigFile name='/etc/inittab' />
    <ConfigFile name='/etc/passwd' />
    <ConfigFile name='/etc/securetty' />
    <ConfigFile name='/etc/shadow' />
    <ConfigFile name='/etc/updatedb.conf' />
    <ConfigFile name='/usr/bin/bu' />
    <Package name='4Suite' />
    ...
  </Group>
</Base>

```

Like the Pkgmgr directory, the Svcmgr directory and the Rules directory, the Base directory can contain multiple files and the appropriate file is chosen based on the Group membership:

Svcmgr/ Directory

```
$ ls Base/
centos-4-x86.xml
fedora-core-4-x86.xml
rhel-as-4-x86.xml
rhel-es-4-x86.xml
rhel-ws-4-x86_64.xml
rhel-ws-4-x86.xml
```

Cfg/ Directory

The Cfg is where all the actual ConfigFiles are kept. If you have a ConfigFile element listed in a Bundle you need to provide Bcfg with the actual configuration file to give to the host. The directory structure under the Cfg/ directory defines where the configuration file should be placed on the host machine relative to the / directory. For example, if you want to manage /etc/motd you would create the directory Cfg/etc/motd/ and then place the /etc/motd file in that directory.

Specialized Configuration Files

Some hosts or Groups might need a different version of /etc/motd or might just need to add or remove lines from it. You can create specialized versions for a particular host by appending .H_<hostname> to the configuration file where <hostname> is the fully qualified hostname. So if foo.example.com needed a special version your motd directory might look like:

```
$ ls Cfg/etc/motd
motd
motd.H_foo.example.com
```

Now when you run Bcfg2 on foo.example.com it will get the specialized version and all other hosts will get the generic version.

If you want specialized versions for a Group you would append .G##_ to the file where ## is a two digit number specifying priority and is the name of the Group. Continuing our example if we wanted to have a special motd for the mail-server Group we would have the following:

```
$ ls Cfg/etc/motd
motd
motd.G01_mail-server
motd.H_foo.example.com
```

Now if a host is a member of the mail-server Group it will get the specialized Group version of motd. Now what happens if foo.example.com is also a member of the mail-server Group? Bcfg2 will use the highest priority, most specific file. So foo.example.com will always get motd.H_foo.example.com. If you have specialized versions for multiple groups, the priorities you assign come in to play. Let's say we have the following:

```
$ ls Cfg/etc/motd
motd
motd.G01_mail-server
motd.G02_web-server
motd.H_foo.example.com
```

If you have a host, not foo.example.com, that is a member of both the mail-server and the web-server Groups, it will get the web-server version because it has a higher priority, 02 instead of 01.

Deltas

Bcfg2 has finer grained control over how to deliver configuration files to a host. Let's say we have a Group named file-server. Members of this group need the exact same `/etc/motd` as all other hosts except they need one line added. We could copy `motd` to `motd.G01_file-server`, add the one line to the Group specific version and be done with it, but we're duplicating data in both files. What happens if we need to update the `motd`? We'll need to remember to update both files then. Here's where deltas come in. A delta is a small change to the base file. There are two types of deltas: cats and diffs. The cat delta simply adds or removes lines from the base file. The diff delta is more powerful since it can take a unified diff and apply it to the base configuration file to create the specialized file. Diff deltas should be used very sparingly.

Cat Files

Continuing our example for cat files, we would first create a file named `motd.G01_file-server.cat`. The `.cat` suffix designates that the file is a cat. We would then edit that file and add the following line:

```
+This is a file server
```

The `+` at the beginning of the file tells Bcfg2 that the line should be appended to end of the file. You can also start a line with `-` to tell Bcfg2 to remove that exact line wherever it might be in the file.

How do we know what base file Bcfg2 will choose to use to apply a delta? The same rules apply as before: Bcfg2 will choose the highest priority, most specific file as the base and then apply deltas in the order of most specific and then increasing in priority. What does this mean in real life. Let's say our machine is a web server, mail server, and file server and we have the following configuration files:

```
motd
motd.G01_web-server
motd.G01_mail-server.cat
motd.G02_file-server.cat
motd.H_foo.example.cat
```

If our machine isn't `foo.example.com` then here's what would happen:

Bcfg2 would choose `motd.G01_web-server` as the base file. It is the most specific base file for this host. Bcfg2 would apply the `motd.G01_mail-server.cat` delta to the `motd.G01_web-server` base file. It is the least specific delta.

Bcfg2 would then apply the `motd.G02_file-server.cat` delta to the result of the delta before it. If our machine is `foo.example.com` then here's what would happen:

Bcfg2 would choose `motd.G01_web-server` as the base file. It is the most specific base file for this host. Bcfg2 would apply the `motd.H_foo.example.com.cat` delta to the `motd.G01_web-server` base file. The reason the other deltas aren't applied to `foo.example.com` is because a `.H_` delta is more specific than a `.G##_` delta. Bcfg2 applies all the deltas at the most specific level.

:info Files

By default Bcfg2 installs configuration files with owner root, group root, and permissions 0644. This may or may not be how you want a particular file installed. For instance, if you want to manage a cron script it needs to be installed so it can be executed. This is where `:info` files come in. For each configuration file you can specify a `:info` file to tell Bcfg2 how to install that particular configuration file. Let's say you have cron script, `foo`, you want installed. You'd first create

etc/cron.d/foo/ and put your script in that directory. Next you would need to create a :info file to tell Bcfg2 to install the file so it is executable:

```
perms: 0755
```

The :info has the format of

```
<parameter1>: <value1>
<parameter2>: <value2>
...
```

The parameters Bcfg2 understands are:

owner

The owner to assign to the file.

group

The group to assign to the file.

perms

The permissions, in octal, to assign to the file.

encoding

How to encode the file in transit to the host. This option is necessary for binary files.

paranoid

Tells Bcfg2 whether or not to make a backup of the file before replacing it.

Let's say you have a library that needs to be installed and owned by user and group nobody and you want it backed up. Your :info would look like:

```
owner: nobody
group: nobody
perms: 0755
encoding: base64
paranoid: true
```

As an alternative, ":info" files can also be named "info".

Bcfg2 Reports

Bcfg2 has the ability to generate reports to make it easier to see at a glance the state of your environment. It can report via three methods: HTML web page, RSS feed, email.

Report Configuration

Report generation requires the etc/report-configuration.xml and looks something like:

```
<Reports>
  <Report name='Example' good='Y' modified='Y'>
    <Delivery mechanism='www' type='nodes-digest'>
      <Destination address='/var/db/bcfg/www/stats.html' />
    </Delivery>
    <Delivery mechanism='rss' type='nodes-individual'>
      <Destination address='/var/db/bcfg/www/stats.rss' />
    </Delivery>
  </Report>
</Reports>
```

```

    <Delivery mechanism='mail' type='nodes-digest'>
      <Destination address='somebody@example.com' />
    </Delivery>

    <Machine name='.*' />
  </Report>
</Reports>

```

This is a good example showcasing several of the report generation features. In this example we define a report named "Example" which includes good as well as bad nodes and those nodes that were modified in the last run. For this report there are three delivery mechanisms: www, rss, mail. The www mechanism is the HTML generation. The other two are pretty self explanatory. The last parameter is which machines to include in this report. This report includes all the machines that Bcfg2 knows about. here you can give a regular expression to narrow down which machines are reported for. Refer to the Bcfg2 documentation for more detail on the configuring reports.

Report Scripts

There are two scripts that must be run to generate reports. These `/usr/sbin/GenerateHostInfo` and `/usr/sbin/StatReports`. `GenerateHostInfo` gathers the data about the given hosts and writes out `etc/hostinfo.xml`. `StatReports` reads the `etc/hostinfo.xml` and generates the reports based on `etc/report-configuration.xml`. `StatReports` should be run after `GenerateHostInfo` and both are run nightly from cron.

Report Interface

For the www and rss mechanisms, you need to define a way to get to the reports generated. Assume reports are stored in `/var/db/bcfg/ www` and we need to define an Apache Alias to access it. `/etc/httpd/conf.d/bcfg2.conf` defines the Apache configuration:

```

<IfModule mod_alias.c>
    Alias /bcfg2 /var/db/bcfg/www
</IfModule>

<Directory "/var/db/bcfg/www">
    Options Indexes FollowSymLinks
    AllowOverride None
    Order deny,allow
    Allow from all
</Directory>

```